

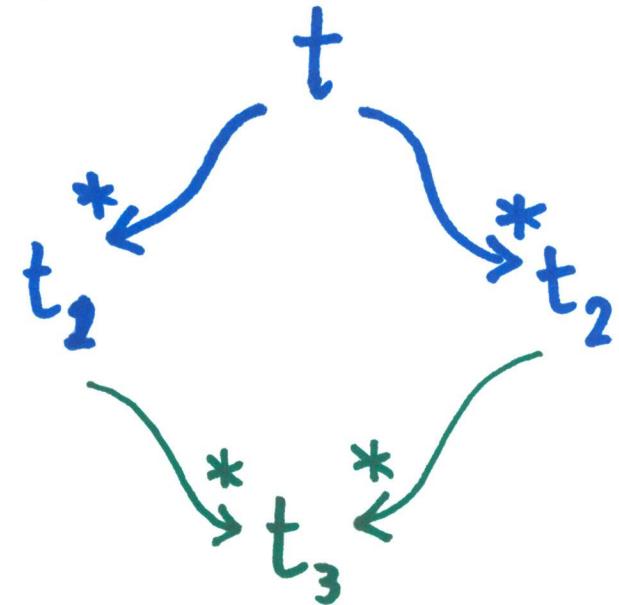
Confluence

It doesn't matter in what order we do reductions

If t is a λ -term and

$t \rightsquigarrow^* t_1$ } I can reduce t
 $t \rightsquigarrow^* t_2$ } to two different terms
 using different reduction sequences

In pictures:



Then there is a term t_3 s.t. Consequence:

$t_1 \rightsquigarrow^* t_3$ } I can find
 $t_2 \rightsquigarrow^* t_3$ } a common reduct

Normal Forms are Unique
 A term can have only one normal form
 (But some terms don't have one)

Evaluation Strategies

for λ -terms

How do we choose which redex to reduce

- Full β -reduction

reduce any redex you want

- Normal order

reduce the leftmost redex
(top redex in AST)

normalizing strategy:

if there is a normal form,
it will find it

- Call-by-name

- If the term is an application
 $(f u)$

reduce f until it becomes
an abstraction

$$f \rightsquigarrow^* \lambda x. t$$

then reduce the main redex

$$(\lambda x. t) u \rightsquigarrow t[x := u]$$

- Never reduce under abstraction
 $\lambda f. t$ is considered
a value

Lazy evaluation (Haskell)

The same with term-graphs
and sharing

• Call-by-value

In an application
 $(f u)$

- reduce f to an abstraction

$$f \rightsquigarrow^* \lambda x. t$$

- reduce u to a value

$$u \rightsquigarrow^* v$$

Then reduce the main
 redex

$$(\lambda x. t) v \rightsquigarrow t[x := v]$$

Idea:

- in call-by-name
 the argument u is passed
 to the function without
 evaluation:
 we use its "name", ie
 its syntactic expression
- in call-by-value
 we evaluate the argument
 to a value
 before passing it
 to the function

Type Systems

Reasons to introduce types

- **Readability**

Human users understand programs better when they have types

- **Correctness**

Types ensure some safety for the input-output relation of programs

- **Outlaw meaningless terms**

Operators can be applied only to arguments of the right type

- **Efficient Compilation**

Operations on a specific data type can be optimized

The need for types

The term $\lambda x. \lambda y. y$

has many meanings:

- Second Projection
- Boolean Value false
- Natural number zero

(by d-eq.: $\lambda f. \lambda x. x$)

We want to distinguish the different uses

General Form of Typing Rules

Two kinds of expressions

terms denote values/objects

types denote sets of values

Typing assertion:

$t : T$

the term t belongs
to the type T

Variables should also have a type

context: assignment of types
to variables

$x:A, y:B, z:C$

Γ

we use capital Greek
letters for contexts

The type of a term t depends
on the type of its free variables

Typing Judgment: $\Gamma \vdash t : T$

Assuming x has type A, y has
type B, z has type C
we can derive that t
has type T

General form of typing rules:

$\Gamma_1 \vdash t_1 : T_1 \quad \dots \quad \Gamma_n \vdash t_n : T_n \quad \downarrow$

$\Gamma_0 \vdash t_0 : T_0$

Then t_0 has type T_0
(Judgments can have different contexts)

Assume the terms
 t_1, \dots, t_n
have these
types

If all judgments have the same context, we can leave it out

$$\frac{t_1:T_1 \dots t_n:T_n}{t_o:T_o}$$

Example: Type system for simple Arithmetic Expressions

Only two types: $T ::= \text{Nat} / \text{Bool}$

Typing Rules:

$$\frac{}{\text{true:Bool}} \quad \frac{}{\text{false:Bool}} \quad \frac{}{\text{zero:Nat}}$$

No Assumptions

$$\frac{t:\text{Nat}}{\text{succ } t:\text{Nat}} \quad \frac{t:\text{Nat}}{\text{pred } t:\text{Nat}}$$

$$\frac{t:\text{Nat}}{\text{iszero } t:\text{Bool}}$$

$$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3:T}$$

any type, but the same for both branches

Meaningless terms cannot be typed:
 $\text{if } (\text{pred false}) \text{ then } (\text{succ true}) \text{ else zero}$
 does not have a type

Full derivation of a typing judgment

zero:Nat

succ zero : Nat

iszero(succ zero) : Bool

zero : Nat

zero : Nat
pred zero : Nat

if (iszero(succ zero)) then zero else (pred zero) : Nat

In this system there are no variables
no need for contexts